Developing high-quality software is tough. ECLAIR is designed to help development, QA, and safety teams reach their quality goals

Coverage of ISO 26262

1 Introduction to ISO 26262:2018

ISO 26262:2018, "Road vehicles — Functional safety," is a series of international functional-safety standards for the automotive industry. It adapts the IEC 61508 series of standards [7] to the functional safety of electrical and/or electronic systems within road vehicles. The first edition of ISO 26262 was published in 2011. The second edition, published in 2018, completely supersedes the previous versions, incorporates a general restructuring of all parts for improved clarity, and contains numerous changes, updates and extensions, among which:

- requirements for motorcycles, trucks, buses, trailers and semi-trailers;
- extension of the vocabulary;
- more detailed objectives and objective oriented confirmation measures;
- management of safety anomalies;
- references to cybersecurity;
- guidance on model based development and software safety analysis;
- guidance on fault tolerance, safety-related special characteristics and software tools.

ISO 26262 provides guidance for the production of *all* software embedded into automotive systems and equipment, whether or not they are safety critical. ISO 26262 approach to risk management is based on the determination of the *Automotive Safety Integrity Level* (ASIL) for each safety function assigned to each subsystem. There are four ASILs: A, B, C and D, with A being the lowest safety integrity level and D being the highest. ASIL D represents likely potential for severely life-threatening or fatal injury in the event of a malfunction and requires the highest level of assurance that the dependent safety goals are sufficient and have been achieved.

In order to determine the ASIL of a safety function, the risk of functional defects has to be evaluated, for each hazardous event, according to three attributes:

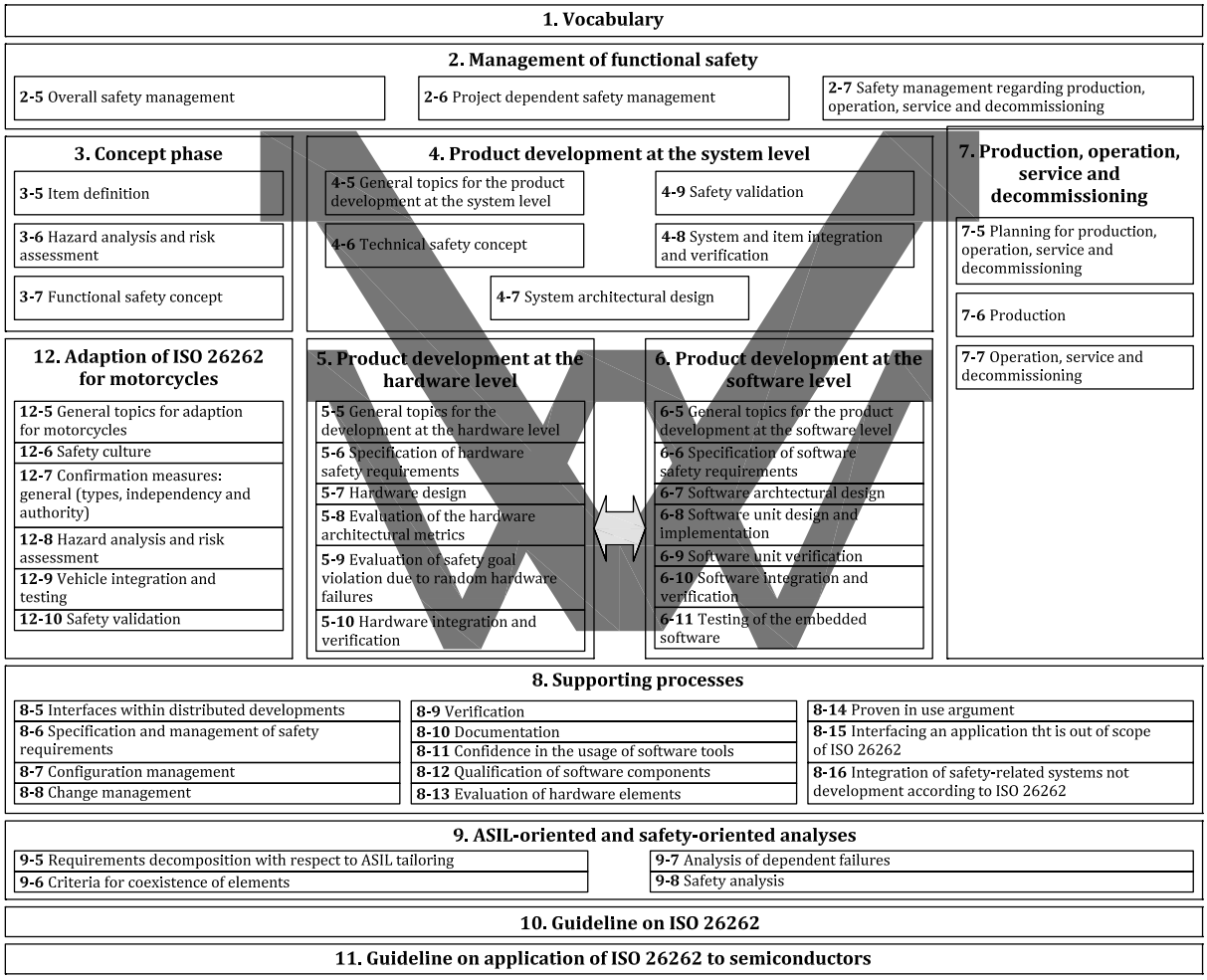
exposure a classification of the probability of the hazardous event (from “incredible” to “high probability”);

severity a classification of its impact on safety (from “No injuries” to “Life-threatening injuries (survival uncertain), fatal injuries”);

controllability a classification of the possibility of the driver and other persons involved in the event, to deal with it (from “Controllable in general” to “Difficult to control or uncontrollable”).

The combination of these attributes determines the ASIL, or that the function is not safety related and thus that there are no requirements to comply with ISO 26262, in which case it is assigned class QM (Quality Management).

ISO 26262 is constituted by 12 parts, which are organized and structured as shown in the following figure:



Overview of the ISO 26262 series of standards

1.1 Role of ECLAIR in Ensuring Compliance with ISO 26262:2018

The ECLAIR Software Verification Platform can be used to comply with several of the objectives of ISO 26262:2018 Part 6 “Product development at the software level” [9] and Part 9 “Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses” [11]. In addition, ECLAIR greatly simplifies compliance with some important prescriptions of ISO 26262:2018 Part 8 “Supporting processes” [10], particularly for Section 11 “Confidence in the use of software tools” (qualification of ECLAIR itself via the *ECLAIR FuSa Pack*, *ECLAIR Qualification Kits*, and the *ECLAIR Qualification Service*,

plus continue compiler qualification via *ECLAIR CerTran*) and Section 12 "Qualification of software components" (via *ECLAIR Scout*).

2 ECLAIR Coverage of ISO 26262:2018 Part 6 Objectives

For automotive applications, Part 6 of ISO 26262:2018 specifies the requirements for product development at the software level [9]. In particular it includes:

- general topics for product development at the software level;
- specification of the software safety requirements;
- software architectural design;
- software unit design and implementation;
- software unit verification;
- software integration and verification; and
- testing of the embedded software.

ISO 26262:2018 Part 6 features several tables defining topics and methods that must be considered in order to comply with the standard. The different topics and methods listed in each table contribute to the level of confidence in achieving compliance with the corresponding requirement. Topics and methods are listed in each table either as *consecutive entries*, numbered with 1, 2, 3, ... in the leftmost table column, or as *alternative entries*, labeled with 1a, 1b, 1c, ... in the same column.

The degree of recommendation to use each topic and method depends on the ASIL, and is symbolically encoded as follows:

++ indicates that the method is highly recommended for the identified ASIL;

+ indicates that the method is recommended for the identified ASIL;

o indicates that the method has no recommendation for or against its usage for the identified ASIL.

For consecutive entries, all listed as highly recommended and recommended topics and methods, in accordance with the ASIL, apply. For alternative entries, an appropriate combination of topics and methods shall be applied in accordance with the ASIL indicated, independent of whether they are listed in the table or not. If methods are listed with different degrees of recommendation for an ASIL, the methods with the higher recommendation should be preferred. A rationale shall be given that the selected combination of methods complies with the corresponding requirement.

The following tables have been obtained by extending the corresponding tables in ISO 26262:2018 Part 6 with a column indicating where ECLAIR, suitably instantiated with the appropriate package, can be used to ensure compliance or to facilitate the achievement of compliance. As ECLAIR provides direct support for MISRA guidelines as well as guidelines from other coding standards, a reference for a guideline should be taken as a reference to the corresponding *ECLAIR service* as described in the *ECLAIR User's Manual*. For example, "MISRA C:2025 Directive 3.1" corresponds to the ECLAIR service `MC4.D3.1`, "MISRA C++:2023 Rule 9.4.1" corresponds to the ECLAIR service `MP2.9.4.1` and "BARR-C:2018 Rule 4.1.a" corresponds to the ECLAIR service `NC3.4.1.a`. For ECLAIR services that do not correspond to published coding standards, the service name is given in teletype font: for example, `B.INDEPENDENCE` is the name of an ECLAIR service that supports automatically enforcing software architectural constraints [2]. A complete definition of all ECLAIR services is contained in the *ECLAIR User's Manual* and, where applicable, in the corresponding coding standard documentation referenced therein.

2.1 MISRA C:2025

MISRA C:2025 [13] is the software development C subset developed by MISRA that is a de facto standard for safety-, life-, security-, and mission-critical embedded applications in many industries, including aerospace, railway, medical, telecommunications and others. MISRA C:2025, which allows coding MISRA-compliant applications in subsets of C90, C99, C11 and C18, is supported, along with all previous versions of MISRA C, by the ECLAIR package called "MC".

2.2 MISRA C++:2023

MISRA C++:2023 [12] is the software development C++ subset developed by MISRA, which is a de facto standard for safety-, life-, and mission-critical embedded applications in many industries including aerospace, railway, medical, telecommunications and others. MISRA C++:2023 completely supersedes MISRA C++:2008 [14], the previous edition of the coding standard, which is still used by many legacy projects. MISRA C++:2023 and MISRA C++:2008 are supported by the ECLAIR package called "MP".

2.3 BARR-C:2018

The *Barr Group's Embedded C Coding Standard*, BARR-C:2018 [5], is, for coding standards used by the embedded system industry, second only in popularity to MISRA C. BARR-C:2018 guidelines include 64 guidelines dealing with language subsetting and project management as well as 79 guidelines concerning programming style. For projects in which a MISRA compliance requirement is not (yet) present, the adoption of BARR-C:2018 is a major improvement with respect to the situation where no coding standards and no static analysis is used. The adoption of the stylistic subset of BARR-C:2018 (79 out of 143 rules) can be part of complying with the MISRA requirement that a consistent programming style is adopted and systematically used as part of the software development process. Moreover, complying with BARR-C:2018, besides avoiding many dangerous bugs, entails compliance with a non-negligible subset of MISRA C:2012 [4]. ECLAIR support for BARR-C:2018 has no equals on the market: it is included in all ECLAIR packages, including the affordable package "B".

2.4 HIS and Other Source Code Metrics

Source code metrics are recognized by many software process standards (and from MISRA) as providing an objective foundation to efficient project and quality management. One well known set of metrics has been defined by HIS (Herstellerinitiative Software, an interest group set up by Audi, BMW, Daimler, Porsche and Volkswagen).

The *HIS source code metrics* [6], while well established, include some metrics that are obsolete and miss others that are required or recommended by software process standards, such as those that allow estimating function coupling. For this reason, ECLAIR supplements HIS source code metrics with numerous other metrics that allow software quality to be assessed in terms of complexity, testability, readability, maintainability and so forth. Keeping track of these metrics also provides an effective and objective method to assess the quality of the software development process. The full set of metrics is available in all ECLAIR packages.

2.5 ISO 26262:2018 *Freedom from Interference*, *Independence*, and *Interference*

ISO 26262:2018 defines *freedom from interference* (FFI) as "absence of *cascading failures* between two or more *elements* that could lead to the violation of a *safety* requirement" [8, Clause 3.65]. Simply put, a *cascading failure* (CF) is a failure that causes an element to fail, which in turn causes a failure in another element [8, Clause 3.17], whereas a *common cause failure* (CCF) is the failure of two or more elements resulting directly from a single specific event (root cause) [8, Clause 3.18]. The union of CFs and CCFs gives what ISO 26262:2018 calls *dependent failures* (DFs), namely, failures that are not statistically independent [8, Clause 3.29]. The notion of DF comes into play in the definition of one

aspect of *independence*: "absence of dependent failures between two or more elements that could lead to the violation of a safety requirement" [8, Clause 3.78].[1] As CFs are a subset of DFs, FFI is instrumental in achieving independence. In turn, achievement of independence or freedom from interference between the software architectural elements can be required because of:

a. the application of an ASIL decomposition at the software level;

b. the implementation of software safety requirements;[2] or

c. required coexistence of the software architectural elements [9, Annex E].

Concerning the last point, criteria for coexistence of elements are given in [11, Clause 6]. When coexistence is required there are two options: (1) all coexisting sub-elements are developed in accordance to the highest ASIL applicable to the sub-elements; (2) the guidance provided in [11, Clause 6] is used to determine whether sub-elements with different ASILs can coexist within the same element. Such guidance is based on the analysis of *interference* of each sub-element with other sub-elements: evidence has to be provided to the effect that there are no CFs from a sub-element with no ASIL assigned (QM), or a lower ASIL assigned, to a sub-element with a higher ASIL assigned, such that these CFs lead to the violation of a safety requirement of the element.[3]

Software partitioning is one of the possibilities for implementing freedom from interference, which must be developed and evaluated taking into account faults concerning *timing and execution*, *memory*, and *exchange of information* [9, Annex D]. ISO 26262:2018 prescribes that software partitioning must be supported, for ASIL D, by dedicated hardware features or equivalent [9, Clause 7.4.9]. A memory protection unit (MPU) is typically used for this purpose; however, as these devices can only enforce partitioning of memory areas and system-on-chip peripherals, other measures are required in order to ensure freedom from interference.

ECLAIR Support for *Freedom from Interference*, *Independence*, and *Interference*

ECLAIR can be used to provide evidence ensuring *freedom from interference*, *independence*, and absence of *interference* as follows:

MISRA compliance: Compliance with the MISRA guidelines reduces the risk of execution blocking due to unexpected excessive loop iterations (one of the issues in the *timing and execution* category) as well as stack overflow (in the *memory* category).

ECLAIR Independence Checker: This is a very general ECLAIR service to detect and check, by control and data flow static analyses, all interactions between user-defined software elements occurring via read or write accesses to shared memory, function calls, passing and returning of data, as well as static dependencies due to header file inclusion and macro expansion.

The `B.INDEPENDENCE` service can be used:

1. In the context of ASIL decomposition applied at the software level, to verify whether the elements implementing the decomposed requirements are sufficiently *independent*.

2. In the implementation of software safety requirements that rely on *freedom from interference* or sufficient *independence* between software components.

[1] This is the technical aspect of *independence*, the other aspect being the organizational one.

[2] E.g., to provide evidence for the effectiveness of monitoring safety mechanisms by showing independence between the monitored element and the monitor.

[3] It is important to realize that "absence of *interference*" and "*freedom from interference*" are distinct concepts in ISO 26262:2018. The latter concept does not depend on ASILs or lack thereof, so that "*freedom from interference*" implies "absence of *interference*," but not the other way around.

3. To determine whether sub-elements with different ASILs can coexist within the same element by verifying absence of *interference* between the sub-elements.

For applications 1 and 2, the user configures the software components by specifying the program elements (functions, variables, ...) that are assumed to be private to each component or shared between components, as well as the allowed interactions between components. ECLAIR will produce a violation message for each unwanted interaction. For application 3, the user configures the ASIL (or QM) of the sub-elements, and the service will flag all program actions whereby a CF might exist from a sub-element with no ASIL assigned (QM), or a lower ASIL assigned, to a sub-element with a higher ASIL assigned. All this greatly simplifies the work to be done in order to ensure compliance with the objectives related to Clause 7 and Annex E of ISO 26262:2018 Part 6, and Clause 6 of ISO 26262:2018 Part 9.

3 ECLAIR Coverage of ISO 26262:2018 Part 8 Objectives

3.1 Section 11 "Confidence in the use of software tools"

For automotive applications, Part 8 of ISO 26262:2018 specifies the requirements for supporting processes [10, Section 11], including:

- the criteria to determine the required level of confidence in software tools;
- the means for the qualification of software tools, in order to create evidence that such tools are suitable to be used to support the activities and tasks required by ISO 26262.

Qualification of ECLAIR

The ECLAIR functionality described above is qualifiable at TCL3 confidence level in compliance with ISO 26262:2018 Part 8. TÜV SÜD audited BUGSENG software development and quality assurance processes for ECLAIR, as well as the internal validation activities performed by BUGSENG on each ECLAIR release. At the end of its assessment, TÜV SÜD awarded BUGSENG the "Software Tool for Safety Related Development" Certificate no. Z10 116151 0001 Rev. 01, attesting that the ECLAIR Software Verification Platform is suitable to be used in safety-related development projects according to ISO 26262:2018 for any ASIL; the requirements of the "Validation of the software tool in accordance with [ISO26262-8, Chapter] 11.4.9" and "Evaluation of the tool development process in accordance with [ISO26262-8, Chapter] 11.4.8" are fulfilled. The ECLAIR FuSa Pack provides all what is necessary to take advantage of this important certification.

The ECLAIR Qualification Kits for ISO 26262 provide further help to safety teams in charge of qualifying ECLAIR for use in safety-related projects where the dependence on the tool operational environment has to be taken into account: the kits contain documents, test suites, procedures and automation facilities that can be used by the customer to independently obtain all the required confidence-building evidence. BUGSENG also offers the ECLAIR Qualification Service, whereby qualified BUGSENG personnel undertakes almost all the qualification effort.

Continuous Compiler Qualification

ECLAIR is also instrumental in achieving *compiler qualification* by validation. This is a service offered in cooperation with our partner *Solid Sands b.v.* as part of their *Compiler Qualification Service*. In case the *SuperTest* compiler test and validation suite reveals defects in the compiler, compliance with functional safety and cybersecurity standards requires appropriate mitigations to be defined and be systematically enforced. Common mitigations include:

1. avoidance of the use of certain language constructs in certain contexts;

2. use of a third party tool to supplement the diagnostic messages not provided by the compiler (e.g., when exceeding translation limits or violating language constraints);

3. avoidance of the use of certain compiler option combinations.

ECLAIR supports all three kinds of mitigations:

1. compliance with the MISRA guidelines excludes the use of several language constructs in certain contexts, and the MISRA guidelines have been designed taking into account the fact that some language constructs are more likely to expose compiler defects than others;

2. ECLAIR checkers for MISRA C:2025 Rule 1.1 and for MISRA C++:2023 Rule 4.1.1 provide suitable diagnostic message for all the violations of the applicable language standard, including the exceeding of translation limits;

3. due to the way ECLAIR works (intercepting all calls to the compiler, linker, assembler and librarian), checks can be defined to ensure the unwanted compiler option combinations are not used.

In addition, ECLAIR's *CerTran* extension automates the configuration of *SuperTest* for compiler qualification by scanning the application build process and creating the exact test configuration files needed to cover all use cases. This can be completely incorporated into a *Continuous Integration system*. Not only does this save a considerable amount of time, but it also avoids configuration errors that can easily occur when scanning the build process manually.

3.2 Section 12 "Qualification of software components"

ECLAIR high configurability makes it an ideal tool when existing code has to be qualified, where extensive *tailoring* of the MISRA guidelines is crucial to the success of the project. Successful tailoring crucially depends on the configurability of the static analysis tool used to check compliance: tailorings that are not supported by the tool are simply unfeasible. ECLAIR has proven its quality in this respect in many projects.[4]

Another feature that is essential when qualifying software components is the precise identification of the component to be qualified [10, Clause 12.4.2.1]. This is particularly challenging in the case of C++, where libraries make extensive use of templates and compile-time evaluation, so that much code has not a direct manifestation in the object code. This problem is solved by the *ECLAIR Scout* product that, given a program, is able to identify precisely which code is actually exercised, even for portions that are evaluated at compile-time.

[4] See, e.g., the paper "Bringing Existing Code into MISRA Compliance," which describes the experience of BUGSENG and AMD on the Xen Hypervisor.

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL				ECLAIR
		A	B	C	D	
1a	Enforcement of low complexity	++	++	++	++	✓[a]
1b	Use of language subsets	++	++	++	++	✓[b]
1c	Enforcement of strong typing	++	++	++	++	✓[c]
1d	Use of defensive implementation techniques	+	+	++	++	✓[d]
1e	Use of well-trusted design principles	+	+	++	++	✓[e]
1f	Use of unambiguous graphical representation	+	++	++	++	–
1g	Use of style guides	+	++	++	++	✓[f]
1h	Use of naming conventions	++	++	++	++	✓[g]
1i	Concurrency aspects	+	+	+	+	✓[h]

[a] HIS [6] and other metrics related to program complexity. ECLAIR allows associating thresholds to each metric.

[b] MISRA C/C++ and BARR-C:2018 define language subsets where the potential of committing possibly dangerous mistakes is reduced.

[c] MISRA C/C++ enforce strong typing on the respective languages. E.g., for MISRA C:2025, Rules 10.1–10.8, 11.1–11.9, and 14.4; for MISRA C++:2023, Rules 7.0.1–7.0.6, 7.11.1 and 8.2.1–8.2.7.

[d] The MISRA C/C++ guidelines promote the use of several defensive programming techniques. E.g., for MISRA C:2025, Directives 4.1, 4.7, 4.11 and 4.14, Rules 14.2, 15.7, 16.4, and 17.7; for MISRA C++:2023, Rules 0.1.2, 9.4.1, 9.4.2, 9.5.1, 9.6.5 and 18.3.1.

[e] The MISRA C/C++ guidelines and thresholds on HIS metrics embody well-trusted design principles.

[f] More than half of the guidelines in BARR-C:2018 [5] concern coding style [3]. MISRA C:2025 Rules 7.3 and 16.5 are also stylistic.

[g] The MISRA C/C++ guidelines provide some minimal naming advice. E.g., for MISRA C:2025, Directives 4.5 and 4.6, Rules 8.3 and 20.2; for MISRA C++:2023, Rules 6.2.2, 6.9.1, 6.9.2 and 19.2.3. More extensive naming advice is included in BARR-C:2018: Rules 4.1.a–d concern module and file names; Rules 5.1.a–c concern type names; Rules 6.1.e–i, 6.4.a and 6.5.b concern function names; Rules 7.1.e–o concern variable names. Two naming rules are also contained in AUTOSAR-C:2009 [1]. In addition, ECLAIR provides configurable naming rules for maximum flexibility.

[h] MISRA C:2025 guidelines regulate concurrency through Directives 5.1–5.3 and Rules 22.11–22.20.

Table 3 — Principles for software architectural design

Methods		ASIL				ECLAIR
		A	B	C	D	
1a	Appropriate hierarchical structure of software components	++	++	++	++	✓[a]
1b	Restricted size and complexity of software components	++	++	++	++	✓[b]
1c	Restricted size of interfaces	+	+	+	++	✓[c]
1d	Strong cohesion within each software component	+	++	++	++	✓[d]
1e	Loose coupling between software components	+	++	++	++	✓[d]
1f	Appropriate scheduling properties	++	++	++	++	–
1g	Restricted use of interrupts	+	+	+	++	–
1h	Appropriate spatial isolation of the software components	+	+	+	++	–
1i	Appropriate management of shared resources	++	++	++	++	✓[e]

[a] ECLAIR provides service `B.INDEPENDENCE` to enforce constraints about layering and to prevent bypassing of software interfaces.

[b] HIS and other metrics related to the size and complexity of software components. ECLAIR allows associating thresholds to each metric.

[c] HIS metrics counting function parameters and MISRA C/C++ guidelines on reduction of variables' scope.

[d] ECLAIR specific metric `B.STFCO_UNIT` allows imposing constraints related to cohesion within and coupling between software components.

[e] ECLAIR service `B.INDEPENDENCE` allows enforcing constraints about the access to shared resources. In addition, management of shared resources is addressed by some MISRA C/C++ guidelines and ECLAIR Bug Finder checks. E.g., for MISRA C:2025, Rules 22.1–22.10; for MISRA C++:2023, Rules 22.4.1 and 30.0.2.

Table 4 — Methods for the verification of the software architectural design

Methods		ASIL				ECLAIR
		A	B	C	D	
1a	Walk-through of the design	++	+	o	o	✓[a]
1b	Inspection of the design	+	++	++	++	✓[a]
1c	Simulation of dynamic behaviour of the design	+	+	+	++	–
1d	Prototype generation	o	o	+	++	–
1e	Formal verification	o	o	+	+	–
1f	Control flow analysis	+	+	++	++	✓[b]
1g	Data flow analysis	+	+	++	++	✓[c]
1h	Scheduling analysis	+	+	++	++	–

[a] ECLAIR analyses of the implemented designs can highlight design defects and facilitate walk-through and inspection.

[b] ECLAIR builds accurate control flow graphs to reason on (feasible and unfeasible) execution paths.

[c] ECLAIR performs a number of data flow analyses to reason about, e.g., pointers, values, and dead stores.

Table 6 — Design principles for software unit design and implementation

Methods		ASIL				ECLAIR
		A	B	C	D	
1a	One entry and one exit point in sub-programmes and functions	++	++	++	++	✓[a]
1b	No dynamic objects or variables, or else online test during their creation	+	++	++	++	✓[b]
1c	Initialization of variables	++	++	++	++	✓[c]
1d	No multiple use of variable names	++	++	++	++	✓[d]
1e	Avoid global variables or else justify their usage	+	+	++	++	✓[e]
1f	Limited use of pointers	+	++	++	++	✓[f]
1g	No implicit type conversions	+	++	++	++	✓[g]
1h	No hidden data flow or control flow	+	++	++	++	✓[h]
1i	No unconditional jumps	++	++	++	++	✓[i]
1j	No recursions	+	+	++	++	✓[j]

[a] MISRA C:2025 Rule 15.5, MISRA C++:2008 Rule 6-6-5 require subprograms to have a single entry and a single exit only. An upper threshold on metric `HIS.RETURN` allows for a more flexible approach.

[b] The MISRA C/C++ guidelines include prescriptions limiting the use of dynamic memory allocation. E.g., for MISRA C:2025, Directive 4.12 and Rules 18.7, 21.3, 22.1 and 22.2; for MISRA C++:2023 Rules 21.6.1–21.6.3.

[c] The MISRA C/C++ guidelines include rules mandating the proper initialization of variables. E.g., for MISRA C:2025, Rules 9.1–9.5; for MISRA C++:2023, Rule 11.6.2.

[d] The MISRA C/C++ guidelines include prescriptions against the multiple use of variable names. E.g., for MISRA C:2025, Rules 5.3, 5.5–5.9 and 21.2; for MISRA C++:2023, Rules 6.4.1 and 5.10.1.

[e] The MISRA C/C++ guidelines include prescriptions against the use of unnecessary global variables. E.g., for MISRA C:2025, Rules 8.7 and 8.9. for MISRA C++:2023, Rule 6.7.2. The specific ECLAIR service `B.GLOBALVAR` allows fine control of acceptable global variables.

[f] The MISRA C/C++ guidelines include rules restricting the use of pointers. E.g., for MISRA C:2025, Rules 8.13, 11.1–11.8, and 18.1–18.5; for MISRA C++:2023, Rules 8.2.3, 8.2.4, 8.2.6–8.2.8, 8.7.1, 8.7.2, 8.9.1 and 10.1.1. The specific ECLAIR services `B.PTRDECL` and `B.PTRUSE` allow fine control of pointers' use.

[g] The MISRA C/C++ guidelines include several rules restricting the use of implicit conversions. E.g., for MISRA C:2025, Rules 10.1, 10.3, 10.4, 10.6, 10.7, 11.1, 11.2, 11.4, 11.5, and 11.9. For MISRA C++:2023, Rules 7.0.1, 7.0.2, 7.0.4–7.0.6, 7.11.1, 8.2.4, 8.2.6, 8.2.7 and 8.3.1.

[h] The MISRA C/C++ guidelines include prescriptions about hidden control flow and data flow. E.g., for MISRA C:2025, Directive 4.9, Rules 2.1, 5.3, 13.2, 15.1–15.7, 16.3, 20.7, 20.9, and 21.4; for MISRA C++:2023, Rules 0.0.1, 6.4.1, 9.3.1, 9.4.1, 9.6.1–9.6.3, 19.0.2, 19.1.3, 19.3.4, 21.10.2 and 28.3.1.

[i] The MISRA C/C++ guidelines include limits on the use of non-structured control-flow constructs as well as other unconditional jumps. E.g., for MISRA C:2025, Rules 14.3, 15.1–15.4, and 21.4; for MISRA C++:2023, Rules 0.0.2, 9.6.1–9.6.3 and 21.10.2. A threshold on metric `HIS.GOTO` allows limiting the use of `goto`.

[j] MISRA C:2025 Rule 17.2 and MISRA C++:2023 Rule 8.2.10 forbid recursion. A threshold on metric `HIS.ap_cg_cycle` also allows ruling out recursion.

Table 7 — Methods for software unit verification

Methods		ASIL				ECLAIR
		A	B	C	D	
1a	Walk-through	++	+	o	o	✓[a]
1b	Pair-programming	+	+	+	+	✓[a]
1c	Inspection	+	++	++	++	✓[a,g]
1d	Semi-formal verification	+	+	++	++	✓[b]
1e	Formal verification	o	o	+	+	–
1f	Control flow analysis	+	+	++	++	✓[c]
1g	Data flow analysis	+	+	++	++	✓[d]
1h	Static code analysis	++	++	++	++	✓[e]
1i	Static analyses based on abstract interpretation	+	+	+	+	✓[f]
1j	Requirements-based test	++	++	++	++	✓[g]
1k	Interface test	++	++	++	++	–
1l	Fault injection test	+	+	+	++	–
1m	Resource usage evaluation	+	+	+	++	–
1n	Back-to-back comparison test between model and code, if applicable	+	+	++	++	–

[a] Compliance to the MISRA C/C++ and the BARR-C:2018 guidelines greatly increases code readability and understandability, thereby facilitating verification activities by walk-through, pair-programming and inspection.

[b] ECLAIR implements numerous verification algorithms based on semi-formal notation.

[c] ECLAIR builds accurate control flow graphs to reason on (feasible and unfeasible) execution paths.

[d] ECLAIR performs a number of data flow analyses to reason about, e.g., pointers, values and dead stores.

[e] All ECLAIR verification algorithms are based on static code analysis.

[f] Several verification algorithms implemented by ECLAIR are formalized in terms of abstract interpretation.

[g] ECLAIR service `B.REQMAN` allows ensuring that all code is forward and backward traceable to documented requirements, including safety requirements. `B.REQMAN` also allows tracing code to the tests and back. The integrated requirements management tool makes ECLAIR a cost-effective, complete solution for requirements-based development and testing.

Table 10 — Methods for verification of software integration

Methods		ASIL				ECLAIR
		A	B	C	D	
1a	Requirements-based test	++	++	++	++	✓[a]
1b	Interface test	++	++	++	++	–
1c	Fault injection test	+	+	++	++	–
1d	Resource usage evaluation	++	++	++	++	–
1e	Back-to-back comparison test between model and code, if applicable	+	+	++	++	–
1f	Verification of the control and data flow	+	+	++	++	✓[b]
1g	Static code analysis	++	++	++	++	✓[c]
1h	Static analyses based on abstract interpretation	+	+	+	+	✓[d]

[a] ECLAIR service `B.REQMAN` allows ensuring that all code is forward and backward traceable to documented requirements, including safety requirements. `B.REQMAN` also allows tracing code to the tests and back. The integrated requirements management tool makes ECLAIR a cost-effective, complete solution for requirements-based development and testing.

[b] ECLAIR executes a number of control flow and data flow analyses.

[c] All ECLAIR verification algorithms are based on static code analysis.

[d] Several verification algorithms implemented by ECLAIR are formalized in terms of abstract interpretation.

4 The Bigger Picture

ECLAIR is very flexible and highly configurable: it supports all kinds of software development workflows and environments.

ECLAIR is fit for use in mission- and safety-critical software projects: it has been designed from the outset to exclude configuration errors that would undermine the significance of the obtained results.

ECLAIR is developed in a rigorous way and carefully checked with extensive internal test suites (tens of thousands of test cases) and industry-standard validation suites.

ECLAIR is based on solid scientific research results and on the best practices of software development.

ECLAIR's unique features and BUGSENG's strong commitment to the customer, allow for a smooth transition to ECLAIR from any other tool.

BUGSENG's quality system has been certified by TÜV Italia (TÜV SÜD Group) to comply with the requirements of UNI EN ISO 9001:2015 for the "Design, development, maintenance and support of tools for software verification and validation" (IAF 33).

BUGSENG is an Arm's Functional Safety Partner, and is thus recognized as a partner who can reliably support their customers with industry leading functional safety products and services.

For More Information

BUGSENG srl
Via Fiorentina 214/C
I-56121 Pisa, Italy
Email: info@bugseng.com
Web: http://bugseng.com

bugseng

**no shortcuts,
no compromises,
no excuses:
software verification done right**

References

[1] AUTOSAR. Specification of C implementation rules. Technical report, AUTOSAR, December 2009.

[2] R. Bagnara, A. Bagnara, and P. M. Hill. Formal verification of software architectural constraints. In DESIGN&ELEKTRONIK, editor, *embedded world Conference 2023 — Proceedings*, pages 271–279, Nuremberg, Germany, 2023. WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany.

[3] R. Bagnara, M. Barr, and P. M. Hill. BARR-C:2018 and MISRA C:2012: Synergy between the two most widely used C coding standards, 2020.

[4] R. Bagnara, M. Barr, and P. M. Hill. BARR-C:2018 and MISRA C:2012 (with Amendment 2): Synergy between the two most widely used C coding standards. In DESIGN&ELEKTRONIK, editor, *embedded world Conference 2021 DIGITAL — Proceedings*, pages 378–391, Nuremberg, Germany, 2021. WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany.

[5] M. Barr. *BARR-C:2018 — Embedded C Coding Standard*. Barr Group, www.barrgroup.com, 2018.

[6] H. Kuder et al. HIS source code metrics. Technical Report HIS-SC-Metriken.1.3.1-e, Herstellerinitiative Software, April 2008. Version 1.3.1.

[7] IEC. *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. IEC, Geneva, Switzerland, April 2010.

[8] ISO. *ISO 26262:2018: Road Vehicles — Functional Safety — Part 1: Vocabulary*. ISO, Geneva, Switzerland, December 2018.

[9] ISO. *ISO 26262:2018: Road Vehicles — Functional Safety — Part 6: Product development at the software level*. ISO, Geneva, Switzerland, December 2018.

[10] ISO. *ISO 26262:2018: Road Vehicles — Functional Safety — Part 8: Supporting processes*. ISO, Geneva, Switzerland, December 2018.

[11] ISO. *ISO 26262:2018: Road Vehicles — Functional Safety — Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses*. ISO, Geneva, Switzerland, December 2018.

[12] MISRA. *MISRA C++:2023 — Guidelines for the use of C++17 in critical systems*. The MISRA Consortium Limited, Norwich, Norfolk, NR3 1RU, UK, October 2023.

[13] MISRA. *MISRA C:2025 — Guidelines for the use of the C language in critical systems*. The MISRA Consortium Limited, Norwich, Norfolk, NR3 1RU, UK, March 2025.

[14] Motor Industry Software Reliability Association. *MISRA C++:2008 — Guidelines for the use of the C++ language in critical systems*. MIRA Limited, Nuneaton, Warwickshire CV10 0TU, UK, June 2008.